CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **11**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Quantitative dynamic-memory analysis for Java[‡]

Diego Garbervetsky[*][†], Sergio Yovine, Víctor Braberman, Martín Rouaux, Alejandro Taboada

Departamento de Computación, FCEyN, UBA

SUMMARY

Space- and time-predictability are hard to achieve for object-oriented languages with automated dynamic-memory management. Although there has been significant work to design APIs, such as the Real-Time Specification for Java (RTSJ), and to implement garbage collectors to enable real-time performance, quantitative space analysis is still in its infancy. This work presents the integration of a series of compile-time analysis techniques to help predicting quantitative memory usage. In particular, we focus on providing tool-assistance for identifying RTSJ scoped-memory regions, their sizes, and overall memory usage. First, the tool-suite synthesizes a memory organization where regions are associated with methods. Second, it infers their sizes in *parametric* closed-form in terms of relevant program variables. Third, it exhibits a parametric upper-bound on the amount of available free memory required to execute a method. The experiments carried out with a RTSJ benchmark, a real-time aircraft collision detector, show that semi-automatic, tool-assisted generation of scoped-based code is both helpful and doable.

KEY WORDS: Java Real-Time; Scoped-Memory; Quantitative Memory Requirements; Static Analysis

1. Introduction

Garbage-collected object-oriented languages, like Java, are gaining momentum for programming real-time applications. An example of this is the use of real-time Java in time-critical financial applications. This poses the challenge of predicting both their execution time and memory consumption. However, this is known to be extremely difficult for systems with automated memory reclaiming.

There has been significant improvement in the development of automatic garbage collectors in order to meet real-time requirements. Examples are Metronome [5], Sun GC based on [23]

[*]Correspondence to: Departamento de Computación, FCEyN, UBA, Pabellón I, Ciudad Universitaria, (C1428EGA) Buenos Aires, Argentina. Tel./Fax:+54-11-4576-3359.
[†]E-mails: diegog@dc.uba.ar, syovine@dc.uba.ar, vbraber@dc.ubar. Researchers at CONICET.

Received
Revised

and JamaicaVM [32], which achieve automatic memory management without compromising time-predictability. However, there is still the problem of predicting the dynamic memory footprint of the program, that is how much memory it requires to execute without throwing an out-of-memory exception. This issue is particularly important in critical systems requiring certification, and in memory-constrained embedded applications. Determining upper-bounds on dynamic memory consumption is very hard due to the difficulty of quantifying the behavior of GCs with respect to application's memory usage.

An interesting approach to gain control on time and space is to change the memory organization model allocating objects in regions [33, 19]. The Real-time Specification for Java (RTSJ) [20] supports application-level region-based memory management through `ScopedMemory` areas. Under suitable conditions, this environment guarantees both predictable-time operations and predictable-space occupancy at the expense of making programming more difficult. Indeed, complying with RTSJ rules complicates reusing legacy code without careful modification [28]. More importantly in practice, it forces the programmer to adopt new coding habits and to reason in a new paradigm quite different from Java. Moreover, ensuring predictability *requires providing upper-bounds of region sizes.*

This paper presents an integration of a series of techniques and tools into a tool-suite to help developers tackling the aforementioned problems. It also explains how to use it to produce sound and predictable scope-memory managed code from conventional Java code. The tool-suite is meant to be used following the scenario depicted in Fig. 1. The basic idea is to follow a two-step approach to assist programmers generate regions. First, infer memory scopes with the aid of escape analysis [31, 30]. Second, fine-tune the memory layout by resorting to a region edition tool [15] and explicit program annotations.

Furthermore, the tool-chain provides two unique key features, which are independent of the way objects are organized into regions. The first one is the ability to produce upper bounds of region sizes as functions of program variables [8]. Such functions are used at runtime to determine the actual sizes of RTSJ memory-scopes. The second one is the symbolic over-approximation of the dynamic memory footprint of a method (including its callees) [7]. This enables checking whether the method can be safely executed without running out of memory. That is, the resulting expression (actually, a polynomial) can be seen both as a *pre-condition*, stating that the method requires that amount of available free memory before executing and, also, as a *certificate*, ensuring that the method will not use more memory than the specified amount.

The use of the tool-chain is illustrated with an RTSJ benchmark: the CDx aircraft collision detector [24].This experiment shows how the tool-chain can assist programmers in several ways, by proposing non-evident memory scopes, enabling manipulation of the memory organization, yielding parametric expressions to appropriately dimension regions at runtime, and providing upper-bound of memory footprint. In particular, this helps analyzing at compile-time the effects of several possible memory organizations, in order to choose the best one in terms of space. This benchmark allows both validating the methodology and showing the current limitations of the tool-suite.

Figure 1. Use scenario.

2. Region Synthesis

The first step consists in analyzing the program so as to provide a scoped-memory organization of the application. Scoped-memory management is based on the idea of allocating objects in *regions* associated with the lifetime of a computation unit, i.e., its scope. A computational unit can be a method, a thread, etc. In this work we will assume programs are single-threaded and non-recursive, and regions are in one-to-one correspondence with methods. This leads to a run-time memory organization in the form of a region stack.

When a method finishes its execution, its objects are automatically collected. There are restrictions on the way objects can reference each other to avoid dangling references. An object o_1 belonging to region r can reference an object o_2 only if: *a)* o_2 belongs to r, or *b)* o_2 belongs to a region that is always active when r is active, or *c)* o_2 is in the heap. An object o_1 cannot point to an object o_2 in region r if: *a)* o_1 is in the heap, or *b)* r is not active at some point during o_1's lifetime.

In order to alleviate the burden and risks of manual operations on regions, these are automatically inferred from the program based on escape analysis [18]. Intuitively, an object *escapes* out of m when its lifetime is longer than m's. Thus, it cannot be safely collected upon m's termination. An object is *captured* by m when its space can be reclaimed when m returns.

It is possible to synthesize by program analysis a memory organization that associates a region to each method m (called m-region) in such a way that scoped-memory restrictions (like RTSJ) are fulfilled by construction. Nevertheless, determining precise object lifetime is undecidable. Therefore, we advocate a semi-automatic approach by making the programmer participate in the analysis and the transformation process. For this, our implementation of the use scenario in Fig. 1 combines an automatic static-analysis tool (e.g., [29]) for *region synthesis*, with a region visualizer and editor, called `JScoper` [15], for *region tunning.*

RUNNING EXAMPLE: Fig. 2 presents the example we will use throughout the paper to illustrate our approach. It computes aggregated information from data produced by an array of input sensors every minute at different locations. Precisely, it calculates: 1) the average temperature per sensor, and 2) the minimum and maximum temperatures for all sensors in given time-windows. `avgSensor` traverses the array of sensors computing the average using the helper method `avgList`. `minMax` computes the min-max temperature interval between two periods. To eliminate noise, a filter object validates whether sensed data is within predefined parameters. Fig. 3 shows the program's call graph. It is annotated with information (invariants and allocation sites) whose use will become clear while we detail the tool-suite. Each node in the call graph is attached the list of control points corresponding to `new` statements in the method's body. □

A *control state* is a path in the call graph. A *creation site* is a path in the call graph ending in a control point corresponding to a `new` statement. For instance the creation site $process.3.avgSensor.11$ represents a path from line 3 (a call statement in method $process$) to line 11 (the new statement in method $avgSensor$). An m-region is a set of creation sites captured by m.

EXAMPLE: The set of creation sites reachable from method $process$ is the following:

$$\{process.2, process.3.avgSensor.11,\ process.3.avgSensor.12,\\ process.3.avgSensor.16,\ process.3.avgSensor.15.avgList.44,\\ process.3.avgSensor.15.avgList.47,\ process.6.minMax.25,\ process.6.minMax30\}$$

Using the region-synthesis analysis of [29], we obtain 3 regions:

$$\begin{aligned} \texttt{process}\text{-region} &= \{process.2, process.3.avgSensor.11, process.3.avgSensor.12,\\ &\quad process.3.avgSensor.16, process.6.minMax.25\}\\ \texttt{avgSensor}\text{-region} &= \{avgSensor.15.avgList.44, avgSensor.15.avgList.47\}\\ \texttt{minMax}\text{-region} &= \{minMax.30\} \end{aligned}$$

□

```
void process(SensorList s,
             int samples) {
 float avg;
 List L1 = avgSensor(s);
 Tuple[] mM = new Tuple[samples];
 for(int i = 1; i<=samples; i++)
  mM[i] = minMax(s,0,i);
 ...
}

List avgSensor(SensorList sl) {
 List LR= new ArrayList();
 Iterator itSL = sl.iterator();
 for(;itSL.hasNext();) {
  Sensor S = (Sensor)itSL.next();
  Float avg = avgList(S.records);
  LR.add(avg);
 }
 return LR;
}

Tuple minMax(SensorList sl,
  int b, int e) {
 float max = -100;
 float min= 100;
 Iterator itSL = sl.iterator();
 for(;itSL.hasNext();) {
  Sensor S = (Sensor)itSL.next();
  for(int j=0;j<e-b; j++) {
   Float val = (Float)S.records.get(j
       +b);
   CO cObj = new CO(val);
   if(cObj.validate()) {
    float v = val.floatValue();
    max = v >max? v: max;
    min = v <min? v: min;
   }
  }
 }
 return new Tuple(min,max);
}

float avgList(List L) {
 int c = 0;
 float s = 0;
 Iterator itL = L.iterator();
 for(;itL.hasNext();) {
  Value val = (Value)itL.next();
  CO cObj = new CO(val);
  if(cObj.verify(L)) {
   s+=val.value();
   c++;
  }
 }
 return (c>0 ? s/c :-1);
}
```

Figure 2. Sensor-data aggregator

Figure 3. Call graph showing local invariant of allocation sites

3. Region sizes

In [8] we have presented a general technique to perform quantitative memory analysis, which is capable of computing the size of an m-region, for any m, provided the set of `new` statements that allocate objects in the m-region are given. The idea is a follows. The size of the m-region is a function of the number of times these `new` statements are executed during a single invocation of m. Such a number can be computed by characterizing the iteration space where each allocating statement belongs to with a linear invariant on relevant method variables[†]. Then, the quantity we are looking for is the number of integer solutions of the invariant, which turns out to be computable and definable in polynomial closed-form [12]. The algorithmic solution of this problem requires putting together several complex techniques.

Invariants. The computation of region sizes relies on having invariants that characterize the set of possible valuations of relevant variables at a specific control point in m's body for a given path in the call graph (i.e. a control state). The tool-suite builds a *control-state* invariant by computing the conjunction of all local invariants along the path in the call graph. *Local* invariants can be either provided by assertions "à la" JML, or computed using static or dynamic analysis. Currently, the prototype uses Daikon for dynamic detection of "likely" invariants by executing the program over a set of test cases. Even if the expressions generated by Daikon have a high probability of being invariants, they might not be. To overcome this, likely invariants have to be manually verified to ensure correctness.

Example: A local invariant at $process.6$ is $\{1 \leq i \leq samples\}$. A control-state invariant at $process.6.minMax.30$ is $\{1 \leq i \leq samples \wedge b = 0 \wedge e = i \wedge 0 \leq k \leq sl.size() \wedge 0 \leq j \leq e - b\}$.
□

Inductive variables. Invariants do not need to predicate about every single variable in the program, but only about a *relevant* subset that properly characterizes the iteration space of a control location. Such a subset of variables, called *inductive*, are those which cannot be assigned the very same value in two different passes at the given control point, except possibly in the case of a non-terminating loop. An invariant that only involves parameters and an inductive subset of variables is called *inductive*. The inductive subset of variables is computed using a conservative dataflow analysis that combines live-variables analysis, augmented with field sensitivity, with loop-inductive analysis [25]. Besides standard `for`- and `while`-loops, the tool currently deals with iterator-based patterns for traversing collections.

Example: A minimal inductive set of variables at $process.6$ is $\{i, samples, s1.size()\}$. At $process.6.minMax.30$ is $\{i, samples, s1.size(), b, e, k, j\}$.
□

Counting visits. To count the number of solutions of a control-state inductive invariant associated with a creation site reachable from m, the tool resorts to the technique presented in [12] and implemented in the Barvinok library [34] which counts the number of solution of a parametric linear invariant. The result is a polynomial on the parameters of m.

[†]Technically, relevant variables are inductive ones.

Example: The number of points in the control-state invariant at $process.6.minMax.30$ is

$$\begin{aligned} & C(samples, sl.size()) = \\ & = \#\{(i,k,e,b,j) | 1 \leq i \leq samples \wedge b = 0 \wedge e = i \wedge 0 \leq k \leq sl.size() \wedge 0 \leq j \leq e - b\} \\ & = \frac{1}{2} sl.size().samples^2 + \frac{1}{2} sl.size().samples \end{aligned}$$

□

Region sizes. Recall that an m-region is characterized by the set of creation sites reachable from m which are captured by it. The region-size function returns the region-size in bytes for a given valuation of the parameters. This function is the sum of the computed polynomials each one multiplied by the size in bytes of the corresponding allocated object[‡]. Such a function allows to cope with region sizes which depend on the calling context and may vary from one call to the other.

Example: Let $N = sl.size()$, $s = samples$, $d = s1[i].records.lenght$, and $\mathcal{S}$ the type size:

$$\begin{aligned} \texttt{regSize}_{process}(N,s,d) &= s.\mathcal{S}(Tuple[]) + \mathcal{S}(List) + \mathcal{S}(SLIterator) + N.\mathcal{S}(Float) \\ & \quad + s.\mathcal{S}(SLIterator) \\ \texttt{regSize}_{avgSensor(N,d)} &= N.d.\mathcal{S}(CO) + N.\mathcal{S}(Iterator) \\ \texttt{regSize}_{minMax(N,b,e)} &= (e-b).N.\mathcal{S}(CO) \end{aligned}$$

For simplicity, hereinafter we omit $\mathcal{S}$ (i.e., `regSize` expresses numbers of objects).

□

4. Dynamic memory footprint

The dynamic memory footprint of a program is the amount it needs to execute without ever throwing an out-of-memory exception. This largely depends on the behavior of the garbage collector. Thus, its general characterization is a very difficult task. Nevertheless, [7] proposes a technique to over-approximate it in the case of region-based memory management. Given a method m we compute a parametric expression that bounds from above its dynamic memory footprint. Notice that this includes the footprint of all the methods it calls. The underlying idea is to leverage on two key obervations. Firstly, the machinery developed in previous sections for inferring regions and their sizes. Secondly the ability given by scoped-memory management to precisely control where and when regions are created and destroyed. Roughly speaking, the technique consists in over-approximating the memory occupied by the largest region stack at any given moment.

Region stacks. In our model, region stacks are associated with paths in the call graph. There are two important facts to take into account. (1) Due to branching, there are some region stack configurations that cannot happen at the same time. (2) For every path π (a control state) in the call graph, there will eventually be corresponding region stacks whose region sizes may

[‡]For arrays the computation is a bit trickier, see [8].

Figure 4. Region stacks. P=*process*, avS=*avgSensor*, aL=*avgList* and mM=*minMax*.

differ in different execution states. Each one will composed of the same regions but instantiated with different sizes depending on the parameter values for each particular calling context.

Example: To illustrate this phenomenon, let us take a look at the running example. Fig. 4 shows the evolution of the region stack for the execution of `process` with two sensors and three samples. The dynamic memory footprint is determined by the maximum region stack size among all the possible instances at runtime. Thus, to approximate it is enough to consider all paths in the call graph starting from `process` and, for each one, sum up the *largest* size an m-region in the stack can get. In Fig. 4 the `minMax`-region varies according to its calling context. The size of this region varies according to the difference between the values of the parameters e and b.
□

Binding invariants. The estimation needs to be expressed in terms of the formal parameters of the root method of the call chain, e.g., `process` in the example above. However, footprints of callees are expressed in terms of their own parameters. Therefore, there we need to bind them in some way. For this, we construct so-called *binding invariants* which are control-state invariants relating the caller's arguments with the callee's formal parameters. That is, a binding invariant models the possible values of variables along the data stack.

Example: A binding invariant for the call of `minMax` from `process` at line 6 is $\{1 \leq i \leq samples \wedge b = 0 \wedge e = i\}$.
□

Maximum region size. Let us denote $\texttt{MaxRegSize}^{\pi.m}_{mua}$ the largest region size of an m-region in a path $\pi.m$ in the call graph rooted at mua. It is defined as follows:

$$\texttt{MaxRegSize}^{\pi.m}_{mua}(P_{mua}) \quad = \quad \text{Maximize } \texttt{regSize}_m(P_m) \text{ subject to } I^{\pi.m}_{mua}(P_{mua}, P_m, W)$$

where P_{mua} and P_m are the parameters of mua and m, resp., and W are the local variables appearing in the methods in π.

Example: Consider the path $process.6.minMax$. This case is interesting because the calling context changes in every call to method `minMax` due to the outer loop in line 5. The binding invariant is $\{1 \leq i \leq s \wedge b = 0 \wedge e = i\}$ which captures the iteration space and the binding of parameters. Applying the formula above, we otain:

$$\begin{aligned}
&\texttt{MaxRegSize}^{process.6.minMax}_{process}(N,d,s) = \\
&\quad = \text{ Maximize } \texttt{regSize}_{minMax}(N,b,e) \text{ subject to } I^{process.6.minMax}_{process}(N,d,s,e,b,i) \\
&\quad = \text{ Maximize } (e-b)N \text{ subject to } \{1 \le i \le s \wedge b = 0 \wedge e = i\} \\
&\quad = sN
\end{aligned}$$

This is consistent with Figure 4 that graphically shows that the largest region is obtained in the last iteration of the loop.
□

Clearly, it will be very expensive to solve this maximization at run-time, but off-line computation at compile-time poses the problem of solving it parametrically. Indeed, this non-trivial polynomial maximization problem can be solved by an approach based on Bernstein expansion for handling parameterized multivariate polynomials over a parametric polyhedral space [13]. Roughly speaking, this technique provides a set of polynomials which bound the original one in the given domain. Such polynomials are defined in terms of the *parameters of the domain*. The solution to our problem is the maximum such polynomial.

Dynamic memory footprint. To compute the footprint for a given method *mua*, we have to sum up its maximum region size and the largest one among the maximum region sizes of all its callees.

EXAMPLE: For method `process`, we have:

$$\begin{aligned}
&\texttt{memFootPrint}_{process}(N,d,s) = \\
&\quad = \texttt{MaxRegSize}^{process}_{process}(N,d,s) \\
&\qquad + \max\{\texttt{memFootPrint}^{process.3.avgSensor}_{process}(N,d,s), \texttt{memFootPrint}^{process.6.minMax}_{process}(N,d,s)\}
\end{aligned}$$

Notice that methods `avgSensor` and `minMax` capture everything allocated by its callees. As a consequence, only `avgSensor` and `minMax` regions are relevant for memory footprint computation since regions size of their callees will be zero. It follows that:

$$\begin{aligned}
\texttt{memFootPrint}^{process.3.avgSensor}_{process}(N,d,s) &= \texttt{MaxRegSize}^{process.3.avgSensor}_{process}(N,d,s) \\
\texttt{memFootPrint}^{process.6.minMax}_{process}(N,d,s)\} &= \texttt{MaxRegSize}^{process.6.minMax}_{process}(N,d,s)
\end{aligned}$$

Therefore,

$$\begin{aligned}
&\texttt{memFootPrint}_{process}(N,d,s) = \\
&\quad = \texttt{MaxRegSize}^{process}_{process}(N,d,s) \\
&\qquad + \max\{\texttt{MaxRegSize}^{process.3.avgSensor}_{process}(N,d,s), \texttt{MaxRegSize}^{process.6.minMax}_{process}(N,d,s)\} \\
&\quad = N + 2s + 2 + \max\{Nd + N, sN\} \\
&\quad = Nd + 2N + 2s + 2 \quad (\text{since } d \ge s)
\end{aligned}$$

□

The general formulation is as follows:

$$\texttt{memFootPrint}^{\pi}_{mua}(P_{mua}) = rsize^{\pi}_{mua}(P_{mua}) + \max_{\pi.l.m \in \Pi_{mua}} \left\{\texttt{memFootPrint}^{\pi.l.m}_{mua}(p_{mua})\right\}$$

Figure 5. Comparing total consumption using inferred and refined regions

where Π_{mua} is the set of paths in the call graph rooted at mua.

Indeed, to fully characterize the dynamic memory footprint of a method mua, we still need to consider the objects that are allocated during its execution but which cannot be collected when it returns. Since the escape property is absorbent, it is enough to consider only objects escaping mua ([7]). Thus, the dynamic memory footprint of mua is defined as follows:

$$\boxed{\texttt{memFootPrint}_{mua}(p_{mua}) = \texttt{memEsc}(mua)(p_{mua}) + \texttt{memFootPrint}^{mua}_{mua}(p_{mua})}$$

EXAMPLE: Fig. 5 shows two runs of the running example using 3 and 4 regions, respectively. □

It is worth mentioning that this approach obtains safe bounds of dynamic memory footprint even for other memory reclaiming mechanisms ([7]). The intermediate region inference generation adds an additional level of over-approximation.

5. Compilation and runtime

Once the programmer is satisfied with the results of the quantitative region analysis, the last step of the use scenario is to generate the scoped-memory based Java code. Currently, our prototype tool-suite (partially) supports the following runtime platforms: **JikesVM**[3], **gcc** with the **rc** region library [19], **RTSJ** and **JITS**.

JikesVM and rc. In order to perform scoped-memory management at program level the tool uses an API enabling manipulation of method-scoped memory regions [18]. This API has constructs to create and destroy m-regions and a mechanism to associate objects to m-regions. Objects have to be identified by IDs, such us the program location where they are created (option used in this work). The programmer indicates (typically at region creation) which are the objects the region is going to allocate, by providing their IDs. At object creation, the object identifies itself by presenting its ID. The memory manager checks whether that ID is registered by at least one m-region. It allocates the object in the last region which registered the object or, by default, in the active region.

We are currently implementing a scoped-based region manager were every method header is annotated with region information using the registering mechanism explained before. The implementation heavily uses the fact that region sizes are provided (because we can infer them) to implement a very simple intra-region allocator.

RTSJ. To generate RTSJ code, the approach is to set up a method wrapper, say `m_wrapper` for each method `m`. The code of `m` is replaced by a call to the wrapper which is responsible for creating a region and a runnable, and entering the region with the runnable:

```
... m_wrapper( ... ) {
    Region m_reg = new Region(m_reg_size(...));
    Runnable m_run = new m_run(this, ...);
    m_region.enter(m_run); ...
}
```

where class `Region` extends RTSJ-class `LTMemory`. The (generated) method `m_reg_size(...)` evaluates the expression computed by the region-size inference module on the parameters of `m`. To allocate an object in a region, `Region` uses the registration mechanism mentioned before. Every `new` in `m` is replaced by a call `r.newInstance(Class.forName(...))` in the `run` method of `m_run`. The mechanism is explained in detail in [18].

JITS. JITS is a software framework dedicated to generation, deployment and execution of low-footprint embedded J2SE Java applications. In [29] JITS is extended to support a regions: bytecode interpreter was modified to keep track of regions, without changing nor annotating the bytecode itself. The class loader was also modified to take into account the metadata computed by the static analysis. This implementation was successfully used to execute an embedded real-time MP3 converter.

6. Experimental evaluation

6.1. Running example

To start with, we summarize the analysis of the running example. Table 6.1 shows region sizes and dynamic memory footprint for 3 different configurations (in number of objects). The

Table I. Results for the running example. $N = sl.size()$, $S = samples$ and $d = s1[i].records.lenght$.

Region	Original	Inferred	Manually Refined
process	$\frac{1}{2}N.S^2 + \frac{1}{2}N.S + 2S + dN + 2N + 2$	$N + 2S + 2$	$N + S + 1$
avgSensor	N/A	$Nd + N + 1$	1
minMax	N/A	$(e - b)N$	$(e - b)N + 1$
avgList	N/A	N/A	$L.size() + 1$
$\texttt{memFootPrint}_{process}$	$\frac{1}{2}N.S^2 + \frac{1}{2}N.S + 3S + dN + 2N + 2$	$Nd + 2N + 2S + 2$	$N + 2 + S$ $+max\{d + 1, SN\}$

first one (Original) is a single region capturing all objects allocated in the scope of method `process`. The second one (Inferred) corresponds to the 3 regions automatically deduced by region inference. The last one is obtained as a refinement of the latter by creating a new region for method `avgList`. This region captures the objects created at line 47 which are put in the same region as the list `L` by our region-inference algorithm. In addition, in this refinement iterator no longer escapes.

6.2. CDx benchmark

CDx (Collision Detector)[§][24] is an open-source RTSJ-compliant application benchmark for real-time Java VMs. The CDx core (about 2Kloc, 47 analyzable news) implements a periodic aircraft collision detector. It consists of two core classes (and other helper classes): `PersistentDetectorScopeEntry`, which synchronizes with the (simulated) environment to store the produced radar frames, and `TransientDetectorScopeEntry`, where the actual collision detection algorithm is executed. There is only one memory region used by the transient detector. Almost all objects within its scope are allocated there, except for a subset allocated in the persistent one.

We begin by computing the size of the transient detector memory region. First, fully automated analysis is beyond the capabilities of the tool-suite because the method `dfsVoxelHashRecurse` is recursive. This problem can be circumvented by replacing it by an iterative bfs-based implementation. Second, we introduce a (complexity) parameter p that upper-bounds the number of iterations. Relating this complexity parameter with program variables would require much deeper analysis. Table II, column 1, shows the obtained (non-trivial) expression in terms of p and the simulator's plane counter (c). This result helps setting the parameter `immortal.Constants.TRANSIENT_DETECTOR_SCOPE_SIZE`.

The second step is to try region inference to split the transient scope to lowering down its footprint. This finds regions for `reduceCollisions`, `lookForCollisions` and `createMotions`. Unfortunately, there is no noticeable gain because quantitative analysis produces an almost identical upper bound (column 2). The reason is that all these regions belong to the largest

[§]`http://adam.lille.inria.fr/soleil/rcd/`

Table II. Results for `TransientDetectorScopeEntry` (in number of objects).

Region	Original	Inferred	Manually Refined
run	$\frac{5}{2}pc^2 + (\frac{21}{2}p + 37)c$ -p+15	c+6	c+6
esc(run)	31c	31c	31c
reduceCollisions	N/A	$11pc + c + 2$	$3pc + 2$
lookForCollisions	N/A	$\frac{1}{2}pc^2 + \frac{3}{2}pc - p + 3$	$p + 3$
createMotions	N/A	1	1
determineCollisionSet	N/A	N/A	$\frac{1}{2}p^2 + \frac{3}{2}p - 2$
voxelHashing	N/A	N/A	$8p + 1$
$\texttt{memFootPrint}_{run}$	$\frac{5}{2}pc^2 + \frac{21}{2}pc - p$ +34c+12	$\frac{1}{2}pc^2 + \frac{25}{2}pc - p$ +33c+5	$max\{3pc + 9p + 6, \frac{1}{2}p^2 + \frac{3}{2}p - 2\}$ $+32c + 6$

region stack. It is indeed possible to manually refine this configuration by creating a new region where to allocate the temporary objects created during voxel hashing. Now, the analysis results in a significant improvement of the footprint (column 3).

We generated RTSJ code for the "inferred" and "manually refined" versions. In both cases, the application run without errors.

6.3. Data-base banking application

The last case study is a real-life, data-base banking application coded in Java, which was observed to spend a lot of time garbage-collecting. It is about 3.4Kloc (600 comments), with around 360 `new` statements. The code is structured as a nesting of calls to methods of the (simplified) form shown in Fig. 6(left). Roughly speaking, `exec` first searches all records in the data-base matching `param` (e.g., SQL select), and then, it either performs some computations using the result, stopping the chain of calls, or, for each match, it creates a new worker which will perform a selection in another table based on a new set of parameters, and so on. The application consists of 6 different instances of `Work`-like classes, namely W-A to W-F (Fig. 6) with a minimum of 10 and a maximum of 222 private instance fields each, and a maximum depth of 6 and breadth of 4 calls.

We were requested by the development company to study its footprint. The real application code is beyond the limitations of our current implementation of the quantitative analysis. Thus, we applied the following methodology. We first ran region synthesis on the original code. This served identifying the regions. In particular, it resulted that every `exec` method could be considered to be *waterproof*, that is, no object created within its lifetime ever escapes its scope. This yielded a total of 18 regions. Moreover, the analysis showed that no newly created object was chained to `exec` parameters. That is, `param` was used as a placeholder for passing values down, while `filter` was used to pass values up. This information was useful to perform a (manual) slicing of the parameters and further simplifications of the code, such as instance field elimination, without changing the behavior with respect to the number of

```
class Work {
 ...
 void exec(Object[] param) {
  Iterator it = select(param);
  if(<cond>) {
   while(it.hasNext()) {
   ...
   }
  }
  else {
   while(it.hasNext()) {
    Object[] filter = new Object[] {
         ... }
    Work w = new Work(filter);
    Object[] p = getParams(it.next()
         );
    w.exec(p);
   }
  }
 }
}
```

Figure 6. Data-base banking application.

Table III. Results for the banking application. m: elements to process. CS: captured creations sites.

Method	CS	MRS	Method	CS	MRS	Method	CS	MRS
WA1-ex	90	$3m^2 + 6m + 81$	WA2-init	3	3	WA2-ex	23	$5m + 18$
WA2-a	6	6	WB1-ex	23	$m + 22$	WC2-init	10	10
WB2-ex	7	$m + 6$	WC2-a2	4	4	WC1-ex	57	57
WC2-a3	5	5	WC2-ex	88	$12m^2 + 46m + 40$	WD2-ex	2	2
WC2-a1	8	$m + 7$	WE1-ex	2	2	WD1-ex	16	$m + 15$
WF2-ex	3	3	WF1-ex	43	$5m + 38$	main	1	1
$\texttt{memalloc}_{main}(m) = m^4 + 32m^3 + 130m^2 + 200m + 103$								
$\texttt{memFootPrint}_{main}(m) = 15m^2 + 59m + 221$								

created objects [¶]. After this transformation, we were able to run the prototype and to obtain in less than 8 minutes the polynomials for each of the 18 regions (Table III): 5m of Daikon for computing local invariants, 50s for region synthesis, and 75s for quantitative analysis. `memalloc` is an upper-bound of the total number of objects allocated by the application. Notice that the dynamic memory footprint for region-based management ($\mathcal{O}(m^2)$) is significantly smaller than `memalloc` ($\mathcal{O}(m^4)$). Hence, by transforming the application into a functionally equivalent region-based one, we can get rid of garbage collecting an important amount of objects.

[¶]Indeed, this simplification was done to ease Daikon's task in finding local invariants. Another option would have been manual tunning of the set of inductive variables.

7. Related Work

Escape analysis and region inference (e.g., [31, 14, 9, 18, 30]), on one hand, and quantitative analysis (e.g., [1, 2, 11, 8, 7, 21]), on the other, are currently subject of intensive research (see [17] for a detailed account). Nevertheless, we are not aware of any other integrated tool-suite, like ours, able to generate scoped-memory regions and method-level certificates of memory usage.

The closest related works are those of Chin et al. [10, 11], Albert et al. [1, 2] and Gulwani [21, 22]. The work of Chin relies on a type system and Presburger arithmetic. It statically checks whether user-provided size annotations hold. Inference of upper-bounds of heap and stack sizes are also possible. Even if their type system allows explicit aliasing and object disposal annotations, it is up to the programmer to state the size constraints. In contrast, our technique supports non-linear size expressions and automated synthesis of memory management information. Albert proposes a parametric cost analysis for sequential Java. The idea is to translate a program into a recursive abstract representation, compute appropriate linear invariants, and then compute the so-called *cost relations*, which are systems of recurrent equations in terms of input parameters. It also allows object deallocation by approximating objects' lifetime using escape analysis. This approach is not limited to polynomials, but it is not meant to perform region synthesis and code generation by program transformation.

Gulawani [21, 22] presents a technique for computing symbolic bounds on the number of statements executed by a method in terms of its scalar inputs and user-defined quantitative functions of input data-structures. The idea is instrumenting the program with counters which facilitates linear invariant generation tool the computation of linear bounds individually. These bounds are then composed together to generate total bounds.

8. Limitations and future work

We have developed a prototype tool-suite for assisting programers to produce scoped-memory based code, together with parametric certificates of dynamic memory consumption, following the use scenario depicted in Fig. 1. The experimental set-up allowed us to empirically evaluate the strengths and weaknesses of the overall theoretical approach. There are still several limitations that need to be addressed.

As already mentioned, the approach does not support Java programs featuring recursive method calls. This could be acceptable for critical real-time embedded applications, but it is an obstacle to handle a broader spectrum of applications. Notice that recursion does not affect automatic region synthesis (i.e. escape analysis supports recursive programs) but only memory footprint computation. This is because our theoretical approach is not currently able to deal with unbounded regions stacks. Solving this issue requires performing fixpoint computations over polynomials. An alternative idea could be to develop recurrence equations, similarly to [1].

We have not thoroughly studied yet how our approach works for multi-threaded Java programs. Since concurrency affects object lifetime, our region synthesis should be revised and adapted accordingly. Nevertheless, region sizes and memory footprint could be computed using our technique in a concurrent setting with scoped memory management where: a) regions can be determined statically, and b) threads do not interfere with each other in terms of dynamic memory usage (e.g., objects have thread-local lifetimes or are immortal). This is indeed the case in the fragment of the CDx example we have analyzed.

The analysis is better suited to deal with programs that operate with arrays or simple collections and integer variables. Accurately dealing with allocations depending on values inside complex data structures is inherently more difficult because typically such structures are recursive.

Improving scalability heavily relies on solving two problems. The first one consists in having precise program invariants over a minimal set of useful variables. This issue is not intrinsic to our approach and typically requires human intervention which makes analysis of real-world applications painful if programmers do not use assertions (which can be proven at compile time using an static checker such as ESCJava [16]). The second one is inherent to our quantitative inference technique which requires computing global information such as control states and control state invariants. To tackle the limitations we are currently developing a new compositional analysis [27], based on computing method summaries. This enables local reasoning of memory consumption since each method is computed by using only its own statements and relying on its callees method summaries. Compositional analysis can be integrated with type-checking based approaches which are better suited for complex recursive data structures[10].

We are working on a technique that uses type-checked annotations for data-container classes (like the ones provided by standard libraries), and relies on our quantitative inference approach for loop-intensive application code on top of these verified classes. These annotations will also enable analysis of allocations made by native methods and the virtual machine which is mandatory for developing an industry-mature toolsuite. Recursion will also naturally fit in this approach.

Our approach permits user intervention in order to produce refined regions which can be more fine-grained comparing to regions inferred by our automatic region synthesis tool. In order to ensure that manually defined regions do not lead to runtime errors, techniques like [4, 6] aiming at checking safe region usage could be applied.

Regarding region-based run-time support we plan to improve our implementation in `Jikes` to leverage on more quantitative information about regions-sizes and peak consumption which we are currently able to generate. Instead of only using our prediction to provide sizes at region creation time, we could also use our prediction about maximum regions sizes for preallocating sets of regions that are created more frequently.

Most garbage collection algorithms are optimized to achieve high-throughput at the expense of occasional pauses that can block user threads for tens to hundreds of milliseconds. Real-time collectors have shown that sub-millisecond pause times can be achieved at the cost of a reduction in throughput [5]. However, they do not offer predictability in terms of space, and actually, make quantitative inference of dynamic memory consumption even more difficult.

In conclusion, we think our approach and tool for automatically generating region-based code together with quantitative certificates of memory usage can help improving the overall development cycle of Java-oriented real-time systems, in particular in the context of RTSJ [20] and PERC PICO [26].

Acknowledgements The authors want to thank Jan Vitek and his group for the insightful observations and suggestions and the anonymous referees for their valuable comments for paper improvement. This work has been partially funded by CONICET, ECOS A06E02 "Japiqay", ANPCyT grants PICT 32440, PICT-PAE 02278 and UBACyTX021.

REFERENCES

1. Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of java bytecode. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2007.
2. Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Live heap space analysis for languages with garbage collection. In *ISMM*, pages 129–138, 2009.
3. B. Alpern, S. Augart, S.M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, et al. The Jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.
4. C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time Java memory management. *Real-Time Systems*, 37(1):1–44, 2007.
5. David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 81–92, New York, NY, USA, 2003. ACM Press.
6. Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, New York, NY, USA, 2003. ACM.
7. Víctor Braberman, Federico Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 141–150, New York, 2008. ACM.
8. Víctor A. Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.
9. S. Cherem and R. Rugina. Region analysis and transformation for Java programs. *ISMM'04*, 2004.
10. W. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *SAS 05*, 2005.
11. W.N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *Proceedings of the International Symposium on Memory Management (ISMM '08)*, pages 151–160, Tucson, Arizona, June 2008.
12. P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *ICS'96*, pages 278–285, 1996.
13. Ph. Clauss and I. Tchoupaeva. A symbolic approach to bernstein expansion for program analysis and optimization. In Evelyn Duesterwald, editor, *13th International Conference on Compiler Construction, CC 2004*, volume 2985 of *LNCS*, pages 120–133. Springer, April 2004.
14. M. Deters and R. K. Cytron. Automated discovery of scoped memory regions for real-time java. In *ISMM 02*, pages 25–35, 2002.
15. Andrés Ferrari, Diego Garbervetsky, Victor Braberman, Pablo Listingart, and Sergio Yovine. Jscoper: Eclipse support for research on scoping and instrumentation for real time java applications. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 50–54, New York, NY, USA, 2005. ACM Press.
16. C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 02*, 2002.
17. Diego Garbervetsky. *Parametric specification of dymamic memory utilization.* PhD thesis, DC, FCEyN, UBA, November 2007.
18. Diego Garbervetsky, Chaker Nakhli, Sergio Yovine, and Hichem Zorgati. Program instrumentation and run-time analysis of scoped memory in Java. *RV 04, ETAPS 2004, ENTCS, Barcelona, Spain*, April 2004.
19. D. Gay and A. Aiken. Language support for regions. In *PLDI 01*, pages 70–80, 2001.
20. James Gosling and Greg Bollella. *The Real-Time Specification for Java.* Addison-Wesley Longman Publishing Co., Inc., 2000.

21. Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 375–385, New York, NY, USA, 2009. ACM.
22. Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 127–139, New York, NY, USA, 2009. ACM.
23. R. Henriksson. Scheduling garbage collection in embedded systems. *PhD. Thesis, Lund Institute of Technology*, 1998.
24. Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. Cdx: a family of real-time java benchmarks. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 41–50, New York, NY, USA, 2009. ACM.
25. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
26. K. Nilsen. Improving abstraction, encapsulation, and performance within mixed-mode real-time Java applications. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 13–22. ACM New York, NY, USA, 2007.
27. Diego Piemonte and Diego Garbervetsky. Prediccion parametrica de requerimientos de memoria.especificacion modular. Master's thesis, Departamento de Computación. FCEyN. UBA., August 2009.
28. Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time Java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC)*, Vienna, Austria, May 2004.
29. Guillaume Salagnac, Christophe Rippert, and Sergio Yovine. Semi-automatic region-based memory management for real-time java embedded systems. In *Proceedings of 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, Aug 2007.
30. Guillaume Salagnac, Sergio Yovine, and Diego Garbervetsky. Fast escape analysis for region-based memory management. *Electronic Notes Theoretical Comput. Sci.*, 131:99–110, 2005.
31. Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP 01*, volume 36, pages 12–23, 2001.
32. Fridtjof Siebert. Hard real-time garbage-collection in the jamaica virtual machine. *rtcsa*, 00:96, 1999.
33. M. Tofte and J.P. Talpin. Region-based memory management. *Information and Computation*, 1997.
34. S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Analytical computation of ehrhart polynomials: enabling more compiler analyses and optimizations. In *CASES '04*, pages 248–258. ACM, 2004.